VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Absolvování individuální odborné praxe

Individual Professional Practice in the Company

2019 Jiří Lagan

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jiří Lagan**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Absolvování individuální odborné praxe
Individual Professional Practice in the Company

Jazyk vypracování: čeština

Zásady pro vypracování:

1. Student vykoná individuální praxi ve firmě: Tieto Czech s.r.o.
2. Struktura závěrečné zprávy:
   a) Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta.
   b) Seznam úkolů zadaných studentovi v průběhu odborné praxe s vyjádřením jejich časové náročnosti.
   c) Zvolený postup řešení zadaných úkolů.
   d) Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe.
   e) Znalosti či dovednosti scházející studentovi v průběhu odborné praxe.
   f) Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení.

Seznam doporučené odborné literatury:

Podle pokynů konzultanta, který vede odbornou praxi studenta.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **doc. Mgr. Jiří Dvorský, Ph.D.**

Konzultant bakalářské práce: Mgr. Zdeněk Dřízga

Datum zadání: 01.09.2018

Datum odevzdání: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 8. dubna 2019 .....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

V Ostravě 8. dubna 2019

Tieto Czech s.r.o.
28. října 3346/91
702 00 Ostrava - Moravská Ostrava
IČO 64608051 DIČ CZ64608051

Zdeněk Dřízga

Rád bych poděkoval společnosti Tieto Czech s.r.o. a Mgr. Zdeňku Dřizgovi, za poskytnutí možnosti absolvování individuální odborné praxe, i mým kolegům z projektu Spirála se kterými jsem spolupracoval během této praxe. Poděkovat bych chtěl také doc. Mgr. Jiřímu Dvorskému, Ph.D. za konzultace, vedení a odbornou pomoc při zpracování této práce.

Abstrakt

V této bakalářské práci je popsána autorova odborná praxe na pozici Software Developer ve společnosti Tieto Czech s.r.o., její průběh a uplatnění dovedností a znalostí nabytých během studia. V úvodu práce představuji společnost Tieto Czech s.r.o. včetně krátké historie. Dále se zabývám svou pozicí na přiřazeném projektu a celkovým průběhem vývoje aplikace. Také představuji projekt Spirála a jeho cíle. Poté popíšu mnou řešené úkoly, na kterých jsem během vývoje pracoval včetně použitých technologií. V závěru práce hodnotím použité a chybějící dovednosti, nově nabyté znalosti i celkový průběh praxe.

Klíčová slova: odborná praxe, webová aplikace, vývoj, Tieto, React, TypeScript, ElasticSearch

Abstract

In this bachelor thesis is described author's professional practice in company Tieto Czech s.r.o as a Software Developer, its course and application of skills and knowledge acquired during my studies. In the beginning of the thesis I introduce the company Tieto Czech s.r.o. and its history. Further then I present my position on project I was assigned to and describe whole process of development of the application. I present the project itself together with its goals. In the next part I will describe tasks I worked on, and I also mention used technologies. In conclusion I evaluate my used skills and missing skills, newly gained knowledge and overall course of the professional practice.

Key Words: professional practise, web application, development, Tieto, React, TypeScript, ElasticSearch

Obsah

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
CSR	–	Corporate Social Responsibility
DML	–	Data Manipulation Language
DOM	–	Document Object Model
ES5	–	EcmaScript 5
ES6	–	EcmaScript 6
HTML	–	Hypertext Markup Language
IDE	–	Integrated Development Environment
JSON	–	JavaScript Object Notation
JSX	–	JavaScript XML
OOP	–	Object Oriented Programming
SEO	–	Search Engine Optimalization
SQL	–	Structured Query Language
SSR	–	Server Side Rendering
TSX	–	TypeScript XML
UI	–	User Interface
URL	–	Uniform Resource Locator
XML	–	Extensible Markup Language

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Při výběru tématu bakalářské práce se mi nejvíce zalíbila možnost absolvovat ji formou odborné praxe v IT firmě. Pro tuto možnost absolvovat odbornou praxi ve firmě Tieto Czech s.r.o. na pozici Software Developera jsem se rozhodl, protože jsem chtěl získat praktické zkušenosti s profesionálním vývojem softwaru, dozvědět se o postupech a technologiích, které se v praxi používají a doplnit tím své teoretické znalosti nabyté během studia. Také mě zajímalo, jak probíhá vývoj softwaru ve velké nadnárodní firmě, která má po světě více než 13 000 zaměstnanců, k čemuž patří každodenní využití angličtiny a v neposlední řadě také vyzkoušet si práci v týmu, jelikož jsem se s tím při studiu nesetkal. Ve společnosti Tieto jsem pracoval na projektu jménem „Dobromila“. Jedná se o zcela novou aplikaci pro organizaci Spirála o.p.s. Je zaměřena na on-line propojování neziskového, veřejného a soukromého sektoru prostřednictvím jednoduché webové aplikace. Tuto aplikaci vyvíjí Tieto Czech s.r.o. v rámci své CSR – společenské odpovědnosti firem. Já společně s dalšími třemi kolegy jsme měli celý vývoj na starost. V této práci představuji firmu a popisuji pozici i projekt na které jsem pracoval, problémy, které jsem řešil během implementace a jejich řešení. Ke konci uvádím své poznatky, shrnuji znalosti a zkušenosti ze studia, které jsem během odborné praxe využil, a jaké znalosti mi naopak chyběly. V úplném závěru práce hodnotím odbornou praxi a její průběh a naplnění mého očekávání.

2 O společnosti Tieto

2.1 Tieto Corporation

Společnost Tieto vznikla roku 1968 s původním názvem Tietotehdas Oy ve finském městě Espoo. Původně firma fungovala především jako počítačové centrum. Vyvíjely a udržovaly se zde systémy pro Union Bank of Finland a pár lesnických společností. V roce 1998 se firma rozšiřující svůj vliv v oblasti telekomunikací rozhodla zkrátit svůj název na Tieto a o rok později se název opět změnil díky spojení se švédským Enator a vzniklo TietoEnator. Po roce 2000 se firma začala globálně rozšiřovat a založila mnoho poboček mimo země severní Evropy, v České republice, Rusku a Indii. V dnešní době Tieto zaměstnává více než 13 000 odborníků z více než 20 zemí po celém světě [1].

2.2 Tieto Czech

Do České republiky se společnost Tieto rozhodla vstoupit v roce 2001 a o tři roky později zde otevřela své vlastní softwarové centrum, které se nachází v Ostravě. V dnešní době, s více než 2500 zaměstnanci, se jedná o největší pobočku Tieta hned po domácím Finsku a Švédsku. Se svým počtem zaměstnanců je Tieto Czech největším zaměstnavatelem v oblasti IT v Moravskoslezském kraji a patří mezi největší IT společnosti v České republice. V roce 2012 bylo z důvodu velkého počtu zaměstnanců, rozmístěných ve čtyřech různých budovách, potřeba firmu přesídlit do nově vybudovaného komplexního sídla v centru Ostravy – Tieto Towers, kde sídlí dodnes.

3 Proč jsem si firmu vybral

Při výběru firmy, ve které absolvuji svou bakalářskou praxi jsem zohledňoval několik kritérií. Hledal jsem firmu, která je již známá a jde v podstatě o zaručenou kvalitu. Jistotu, že mi firma umožní se učit od jejích profesionálů, ale také se sám rozvíjet a možnost navázat možnou budoucí spolupráci po úspěšném dokončení bakalářské praxe a bakalářského studia. Další kritérium byly technologie používané na praxi. Měl jsem zájem o vývoj webových aplikací, takže se mi hledání omezilo na pár firem a desítku nabídek praxí. Pro prozkoumání všech zbývajících jsem zvolil výše popsanou firmu Tieto Czech s.r.o. a praxi zaměřenou na vývoj zcela nové webové aplikace pro neziskovou organizaci.

Nastoupil jsem tedy do firmy Tieto Czech s.r.o. pod vedením Mgr. Zdeňka Dřizgy. Jeho oddělení se skládá převážně z programátorů starajících se o vývoj a údržbu softwaru. Nachází se tak zde mnoho zkušených programátorů, ale i studentů na stáži pracujících na interních projektech.

3.1 Pracovní pozice

Před nástupem do firmy jsem musel projít jako všichni ostatní zájemci klasickým přijímacím řízením, které se skládalo ze tří kol. V prvním kole jsem se setkal s pracovnicí z oddělení Human Resources a předmětem schůzky byly především mé dosavadní zkušenosti s IT, plány ohledně dalšího studia, a kratší konverzace v anglickém jazyce. To z důvodu, že Tieto je mezinárodní firma, tudíž komunikace probíhá až na výjimky v angličtině, stejně jako veškerá dokumentace, aplikace a manuály jsou psány anglicky.

Ve druhém kole jsem se již setkal s mým budoucím manažerem Mgr. Zdeňkem Dřizgou a jedním programátorem z jeho oddělení. Proběhla diskuze na téma vývoj software, vlastní projekty, a hlavně zkušenosti s vývojem v Javě a JavaScriptu, jelikož to jsou hlavní technologie, které se budou při vývoji nové aplikace používat. Pohovorem jsem úspěšně prošel a byl pozván na závěrečné třetí kolo, na kterém mi již byla nabídnuta pozice Software Developera, kterou jsem s radostí přijal.

4 O projektu Dobromila

Tématem mé praxe byl kompletní vývoj nové webové aplikace, která by měla sloužit jako internetové tržiště pro vzájemnou spolupráci pomáhajících společensky prospěšných organizací a firem i jednotlivců, kteří je chtějí podporovat. Mému týmu byl tak přidělen právě tento projekt. Náš tým se skládal ze čtyř lidí, kteří měl kompletní vývoj této aplikace na starost. Tým se skládal z dalších třech studentů, kteří byli v Tietu na dlouhodobé stáži, já byl jedním z programátorů, kteří se budou starat o frontend část projektu, zbylí dva měli na starost backend aplikace. Před samotným vývojem, jelikož jsme začínali úplně od základu bylo nutné se se zákazníkem – organizací Spirála, sejít a domluvit se, co od aplikace očekávají, co by vlastně měla aplikace umět, a kdo a jak ji bude používat. Po tomto meetingu jsme měli hromadu informací, které bylo třeba řádně a pečlivě projít a zpracovat do podoby požadavků, které nám později pomohly se správným návrhem a výběrem technologií.

5 Aplikace Dobromila

Webová aplikace Dobromila, tvořená pro obecně prospěšnou společnost Spirála bude jakýmsi internetovým tržištěm. Jejím cílem je vytvořit platformu pro nabídku a poptávku služeb neziskových organizací. Na tyto služby budou moct reagovat firmy či samostatné fyzické osoby, které chtějí také pomáhat, ale kontaktovat se mohou i neziskové organizace navzájem. Aplikaci můžeme rozdělit do tří částí. Části aplikace jsou od sebe oddělené pomocí rolí a povolení, které jsou přiděleny každému uživateli v době vytvoření profilu. Jakmile je účtu přidělena role, už ji nelze změnit.

5.1 Část první - Neregistrovaný uživatel

První část slouží uživatelům, kteří nejsou v aplikaci registrovaní. Takovým uživatelům je umožněn volný pohyb po aplikaci, včetně vyhledávání nabídek a poptávek, prohlížení uživatelských profilů i detailů produktů. Takovýto uživatel však nemůže na nabídku či poptávku, která ho zaujala nijak reagovat, ale je vyzván k registraci. Neregistrovaným uživatelům je kromě kontaktování autora produktu znemožněno i vytváření nových nabídek či poptávek.

Pokud má uživatel zájem se do aplikace připojit, může se jednoduše registrovat. Na výběr jsou tři různé druhy profilů, a to Společensky prospěšná organizace, firma nebo fyzická osoba. Podle vybraného druhu profilu jsou po uživateli vyžadovány registrační údaje. Při registraci je nutné zadat kontaktní e-mail, uživatel se pomocí tohoto mailu bude přihlašovat. Po úspěšném odeslání registračního formuláře je na tento e-mail odeslán vygenerovaný ověřovací registrační email s odkazem. Dokud uživatel svůj účet tímto odkazem neaktivuje, není možné se přihlásit.

5.2 Část druhá - Registrovaný uživatel

Po úspěšné registraci a potvrzení registrace kliknutím na odkaz v emailu se uživatel může do aplikace přihlásit. Uživatel po registraci musí projít schvalovacím procesem, kdy je jeho účet a všechny vyplněné informace ověřeny administrátorem. Do té doby mu však není umožněno kontaktovat ostatní uživatele ohledně jejich produktů, či sám produkty vytvářet. Přihlášenému uživateli také přibude v navigaci nové tlačítko administrace, díky kterému se do této sekce může odkudkoliv z aplikace snadno přesunout.

Administrace je rozdělena do několika sekcí. První z nich je správa. Zde uživatel vidí všechny své vytvořené nabídky a poptávky, může je zde upravovat či měnit jejich status. Také jsou zde sekce se žádostmi, kde uživatel vidí jednak žádosti uživatelů o jeho produktu, tak i své žádosti, které poslal jiným uživatelům. Další sekcí je nastavení profilu. Zde může uživatel upravit veškeré údaje o svém profilu, doplnit například popis organizace či firmy, kterou zastupuje, a může také nahrát svůj profilový obrázek. Jakákoliv změna v profilu však není v aplikaci ostatním uživatelům viditelná do doby, než je aktualizace profilu schválena administrátorem. Změny však mohou být také zamítnuty. V takovém případě se profil vrátí do původní podoby před aktua-

lizací. Poslední sekce je určená k vytváření nových nabídek a poptávek. Pomocí jednoduchého formuláře je uživatel proveden tvorbou nového produktu, který lze doplnit o fotografie. Oproti neregistrovaným uživatelům je těmto uživatelům také umožněna reakce na produkty ostatních uživatelů a možnost takto komunikovat přímo s tvůrcem nabídky či poptávky.

5.3 Část třetí - Administrátor

Uživatel v roli administrátora musí být do aplikace přidán ručně, žádná registrace nebo vložení pomocí aplikace není kvůli bezpečnosti možná. Administrátor se může v aplikaci pohybovat jako kterýkoliv jiný uživatel. Samozřejmě nemá možnost reagovat na nabídky či poptávky, ani je nemůže sám vytvářet. Jeho funkce v aplikaci je především dohlížet na uživatele a na jejich produkty, zda nejsou porušována pravidla používání aplikace, či se zde neobjevují nevhodné příspěvky. Dále spravuje celou aplikaci a vyřizuje požadavky uživatelů.

Sekce pro administraci se značně liší od sekce klasického uživatele. První stránka je zaměřena na administraci všech nabídek a poptávek v aplikaci. Administrátor zde může rychle procházet všechny produkty, prohlížet jejich detaily, a v případě, že porušují podmínky je může jednoduše zamítnout, čímž se s okamžitou platností přestanou zobrazovat v celé aplikaci. Druhá stránka je určena ke schvalování účtů. Všechny nově vytvořené účty, i účty které už v aplikaci jsou, ale byly pouze aktualizované, musí projít schválením. Administrátor má možnost zobrazit si detail profilu, kde vidí typ profilu a veškeré údaje zadané při registraci nebo aktualizaci, je schopen si ověřit například zadané IČ společnosti a změny jedním kliknutím schválit či zamítnout. Změny v profilu jsou v okamžiku schválení administrátorem aktualizovány a zveřejněny všem uživatelům.

6 Řešené úlohy

6.1 Časová náročnost

Praxe ve firmě měla trvat nejméně 50 dní, v zimním semestru jsem odpracoval 34 dnů, v letním 25 dnů. Odhad práce byl asi 40 dní, skutečná doba se lehce lišila, jak je možné vidět v tabulce níže. V tabulce není započítán čas strávený komunikací se zákazníkem a kompletním návrhem systému, což zabralo přibližně 14 dnů.

Úloha	Odhadovaný čas	Strávený čas
první	7 dnů	10 dnů
druhá	12 dnů	14 dnů
třetí	14 dnů	12 dnů
čtvrtá	5 dnů	5 dnů

6.2 Úloha první – Implementace komponenty Modal

6.2.1 Zadání úlohy

Mým prvním úkolem byla implementace komponenty Modal. Komponenta bude sloužit k vykreslení modalového dialogového okna zobrazující nějaký obsah. Důraz u vytváření komponent je kladen především na jejich znovupoužitelnost v aplikaci.

6.2.2 Rozbor problému

Před samotnou implementací jsem musel projít mnoho stránek dokumentace knihovny React [2], seznámit se jak se základními, tak i s pokročilejšími technikami v Reactu a pochopit, jak samotný React vlastně funguje. Vykreslení samotné komponenty bylo třeba provést mimo aktuálního rodiče dané komponenty z důvodu neovlivnění dialogu všemi styly aplikovanými na obsah výše.

6.2.3 Implementace

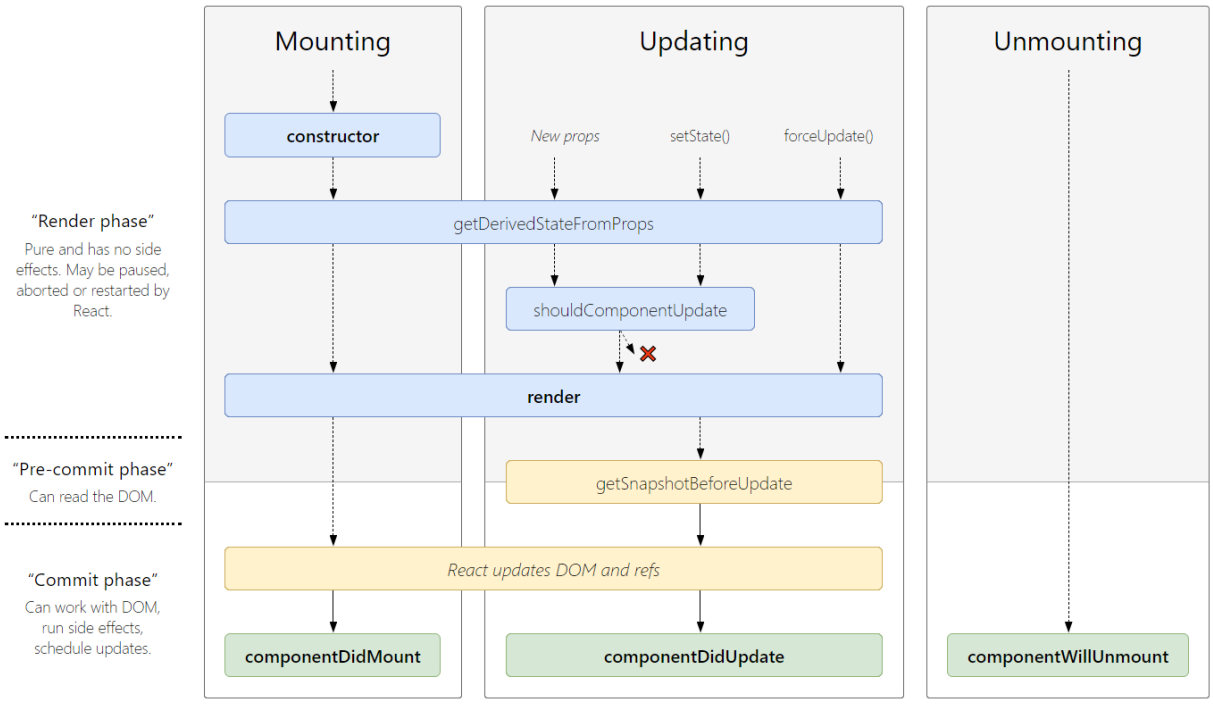
Kód psaný v knihovně React nápadně připomíná zápis XML či HTML tagů přímo do kódu jazyka JavaScript – v mém případě píšu v jazyce TypeScript [3]. Tomuto kódu se říká JSX, pro TypeScript jde o TSX. TypeScript je superset (nadstavba) jazyka JavaScript, který ho rozšiřuje o statické typování, třídy, rozhraní a další věci známé z OOP. Jelikož ale TypeScript nabízí všechny tyto funkce navíc, musím s jazykem TypeScript použít i tzv. transpiler. Transpiler je druh kompilátoru, který za mě přeloží kód napsaný v jednom jazyce, do plně funkčního kódu v jazyce druhém, v mém případě z jazyka TypeScript do jazyka JavaScript. React má velice obsáhlou a často na mnoha příkladech vysvětlenou dokumentaci, takže jsem rychle pochopil, jak React na pozadí funguje. Vykreslování v knihovně React stojí na jediné jednoduché myšlence. Při každém vykreslení se musí nejprve vytvořit reprezentace DOM – React používá tzv. virtuální

DOM, což je strom reflektující strukturu reálného DOM. React poté zjistí změny ve virtuálním DOM, změny následně převede na seznam, který se aplikuje na reálný DOM. Tato implementace umožňuje knihovně React vykreslit pouze změny v DOM. Při změně jednoho textu tak není nutné vykreslovat celou stránku znovu, stačí pouze vykreslit změněnou část, což je extrémně rychlé a efektivní.

Pokud chci v knihovně React vykreslit nějaký element, slouží k tomu metoda `render`. Ta musí vracet jediný element. Tento element však může obsahovat vnořené další elementy, zanoření není omezeno, logicky však s rostoucím zanořením roste složitost kódu a snižuje se jeho čitelnost. React umožňuje posílání dat pouze jedním směrem – tzv. unidirectional data flow. Data tak mohou cestovat pouze směrem dolů, rodič může poslat data svému potomkovi, naopak data poslat nelze. Hledání chyb v takovém prostředí je proto mnohem jednodušší.

Každá komponenta v knihovně React prochází svým životním cyklem. Ke každé části se váže několik metod, jejich volání zajištěno automaticky uvnitř knihovny React. Cyklus je rozdělen do tří částí:

- Mounting – vytvoření komponenty,
- Update – změna dat v komponentě,
- Unmounting – zánik komponenty.



Obrázek 1: Metody životního cyklu v knihovně React [4]

Začal jsem vytvořením souboru index.tsx v adresáři Modal. Poté jsem v tomto souboru vytvořil třídu Modal dědící ze třídy `React.PureComponent` [5]. Pro uložení stavu uvnitř komponenty

lze využít dvou míst. Tím prvním je `state` komponenty. Provádí se metodou `this.setState` a pokud chci k datům opět přistoupit, tak pomocí `this.state`. Hlavní nevýhodou tohoto řešení je, že se k datům komponenty nedá dostat mimo komponentu. Druhým řešením, je uložení těchto dat do `store`, pomocí knihovny React-Redux [6] a následným předáním těchto dat komponentě do properties - `props`. Pokud ale data nejsou potřeba mimo komponentu, je lepší zvolit `state`. Ten jsem také zvolil pro mou komponentu. Jediná proměnná, kterou jsem prozatím potřeboval uložit je proměnná `show` logického datového typu boolean. Ta bude sloužit k určení, zda má být dialog vykreslen či nikoliv.

Při vytvoření komponenty se jako první zavolá její konstruktor. Pokud není konstruktor definován, použije se implicitní konstruktor. Rozhodl jsem se vlastní konstruktor implementovat a nastavil si v něm proměnnou `show` na `false`. Po konstruktoru se volá `componentWillMount`. Tato metoda je volána těsně před tím, než je komponenta připojena do DOM, nebo je zavolána `render` metoda. Najít využití pro tuto metodu může být v mnoha komponentách složité. Jelikož je metoda volaná těsně před `render` metodou, je možné zde vložit například konfiguraci komponenty, ale tu obvykle řeší konstruktor. A jelikož se mezi těmito dvěma metodami nic nestane, je zbytečné nastavovat konfiguraci v této metodě znovu . Také doposud nebyla volána `render` metoda, tudíž komponenta nebyla zatím vložena do DOM. Nejčastěji je tato metoda implementována v kontejnerech, které poté data posílají svým komponentám. Metodu jsem tak proto ani já neimplementoval.

Nejpoužívanější metoda z celého životního cyklu je metoda `render`. To je z důvodu, že metoda `render` je jako jediná ze všech metod životního cyklu povinná. Jak již její jméno napovídá, stará se o vykreslení komponenty do DOM, lépe řečeno do samotného UI. Komponenta může být aktualizována dvěma způsoby, buď úpravou vlastního `state`, či změnou `props`, které jsou do ní shora předávány rodičem. `Render` metoda musí být tzv. pure. To znamená, že nesmí obsahovat žádné vedlejší efekty a musí pokaždé vrátit stejný výsledek pro stejný vstup. Tudíž se v této metodě nesmí upravovat stav komponenty, protože by došlo k jejímu zacyklení. Dalším z důvodů je také to, že `render` metoda by měla být co nejjednodušší. Implementaci této metody jsem si však nechal až na závěr.

Další metodou je `componentDidMount`. Tato metoda je volána ihned potom co je komponenta vložena do DOM a je připravena. Narozdíl od metody `render` je zde dovoleno měnit stav komponenty, a tudíž je to vhodné místo k jakémukoliv nastavení, ke kterému je potřeba aby již byla komponenta v DOM. V této metodě jsem implementoval vytvoření portálu pro komponentu Modal. Pro vytvoření portálu bylo třeba využít knihovny React-DOM [7]. Konkrétně se jedná se o metodu `findDOMNode`, která přijímá jediný parametr a tím je DOM element. Pomocí metody `getElementById` jsem si vyhledal element s id '`portal`', který jsem umístil do hlavního HTML souboru projektu do sekce body, do kterého se mimo jiné vkládá taky celý obsah naší aplikace. Elementem byl prázdný HTML div s nastaveným id, do kterého budu později vkládat komponentu Modal, jelikož chci zajistit její vykreslení mimo svého rodiče. Privátní proměnou `portal` jsem si proto nastavil na tento element. Poté jsem přešel na implementaci metody `render`.

Aby bylo možné v metodě `render` vracet jednotlivé části, které budou vracet samostatné renderovací metody, musím všechny tyto části zaobalit do jediného rodičovského elementu, který vrátí metoda `render`. Jelikož je ale tento krok nutný pouze pro samotný React, a protože nechci zbytečně zatěžovat DOM dalším elementem, je možné využít komponentu zvanou React.Fragment, která mi umožní vrátit více elementů v `render` metodě bez vytváření rodičovského prvku zatěžující DOM [8]. Metoda `render` tak vracela `Fragment` obalující dvě metody starající se o vykreslení jednotlivých částí. V metodě jsem využil `state` komponenty uchovávající informaci o tom, zda je, či není viditelná. Komponenta Modal počítá s tím, že bude spouštěna vždy s nějakou akcí, nejpravděpodobněji se stiskem tlačítka. `Render` metoda tedy vrací `Fragment`, uvnitř kterého jsem volal metodu `renderModalTriggerElement` a poté jsem pomocí ternárního operátoru určil na základě hodnoty `this.state.show`, zda se má, či nemá dialog vykreslit. Metoda `renderModal` však již byla trochu složitější, jelikož vyžadovala vytvoření samotného portálu, který mi umožnil vykreslovat mimo rodičovský prvek. Ten se vytvoří pomocí metody `ReactDOM.createPortal` a má dva parametry. Prvním je komponenta, která se má pomocí portálu vykreslit, a druhý z nich je místo, ve kterém se má vykreslit.

```
...
public renderModal() {
   return ReactDOM.createPortal(this.renderModalComponent(), this.portal);
}

public render() {
   return (
         <React.Fragment>
            {this.renderModalTriggerElement()}
            {this.state.show ? this.renderModal() : null}
         </React.Fragment>
   );
}
...
```

Výpis 1: Implementace portálu v Reactu

Další metoda životního cyklu `componentDidUpdate` je volána ihned po aktualizaci. Je to vhodné místo pro volání metod zajišťující aktualizaci dat. Umožňuje také volání `setState`, které je ale nutné podmínit ověřením předchozích dat, jinak by došlo k vytvoření nekonečné smyčky. Poslední metoda, volaná těsně před odstraněním komponenty z DOM je `componentWillUnmount`. Jelikož již nebude komponenta znovu renderována, je zakázáno jakkoliv upravovat její stav, je však možné provést úklid po komponentě, například odstranění listenerů na klávesy. Implementace komponenty nevyžadovala využití ani jedné z těchto dvou metod životního cyklu.

6.3 Úloha druhá – Implementace získaní dat stránky s produktem

6.3.1 Zadání úlohy

Mým dalším úkolem bylo naimplementovat získání dat pro stránku, která bude zobrazovat jeden produkt včetně všech jeho detailů. Data budu získávat z REST API našeho backendu a před uložením do vnitřního úložiště aplikace je bude třeba ještě přemapovat na imutabilní objekty, které jsou lepší pro manipulaci, případně data pomocí selektorů seřadit a nachystat do formátu vhodného pro zobrazení uživatelům.

6.3.2 Rozbor problému

Jelikož našemu portálu chceme zajistit kvalitní SEO (Search engine optimization) a také co nejlepší zážitek z užívání aplikace pro uživatele, musel jsem na této stránce zobrazovat obsah pomocí technologie zvané Server Side Rendering - SSR. Data jsem tedy musel získávat již na serveru pomocí HTTP požadavků na REST API backendu. Data jsou ale po získání převedena do imutabilních struktur a uložena, proto jsem musel implementovat i tzv. rehydrátory - funkce, které se postarají o obnovení (rehydratovaní) úložiště na straně klienta, jelikož data se při přenášení po sítí musejí převést do formátu JSON, který tyto imutabilní struktury nepodporuje. To je nutné udělat z důvodu, že obsah vyrenderovaný na straně klienta musí přesně odpovídat obsahu, jaký byl vytvořen na serveru, včetně všech dat v úložišti.

6.3.3 Implementace

Jelikož jsem začínal od úplného základu, jako první jsem si vytvořil v projektu složku, ve které budu implementovat vše potřebné, jedná se vlastně o kontejner zastřešující celou stránku, tudíž složka bude pojmenovaná `SingleItemPage`. V adresáři jsem si založil tři prázdné soubory:

- api.ts – soubor sloužící k vytvoření tzv. procesorů - funkce, starající se o definování HTTP požadavků, jejich provedení a uložení odpovědi do vnitřního úložiště aplikace.
- index.tsx – hlavní soubor sloužící k definování kontejneru a veškeré logiky vykreslování dat, včetně samotných komponent.
- selectors.ts – soubor sloužící k definování tzv. selektorů – funkcí, které nám vytáhnou a zpřístupní data z globálního úložiště.

Abych zajistil stahování dat již na serveru, a ne až na straně klienta po otevření a načtení stránky, bylo nutné využít technologii SSR. Data tedy budou stažena přímo na serveru a klientovi pošlu kompletní HTML stránku. Pro zajištění správné funkce SSR je samozřejmě potřeba mnohem více konfigurace a relativně hodně zkušeností s technologií Webpack. Webpack je nástroj, který využívají moderní webové aplikace k optimalizaci výsledného balíku. Jelikož jsem s technologií Webpack nesetkal při žádném vlastním ani školním projektu, byla tato konfigurace přidělena zkušenějšímu kolegovi a já se mohl zaměřit na samotnou implementaci.

Nejprve jsem vytvořil novou třídu `SingleItemPage` která rozšiřuje již knihovnou React implementovanou třídu `React.PureComponent`. Do této rodičovské třídy jsem ještě předal informaci o `props` mého kontejneru. Tento objekt `props` je v podstatě interface, který říká, jaké metody má kontejner dostupné a poslouží mi jak k jednodušší implementaci, tak i jako přidaná kontrola ze strany IDE. To za mě nyní bude hlídat, jestli nepřistupuji k metodám či proměnným, které nejsou dostupné, nebo jsou jiného typu, než je očekáváno.

```
interface SingleItemPageProps {
  router: any;
  fetchData: ({ type, itemId }) => any;
  fetchAccountStats: ({ accountId }) => any;
  creatorStatsSelector: any;
  creatorStatsLoadingSelector: boolean;
  itemSelector: SingleItemType;
  itemImagesItemsSelector: ImageType[];
}

class SingleItemPage extends React.PureComponent<SingleItemPageProps> {
...
}
```

Výpis 2: Definice třídy SingleItemPage

Díky rozšíření třídy `React.PureComponent` mám k dispozici několik metod, které mi pomohli v mém úkolu. Jediná metoda, která se volá přímo na serveru je metoda `componentWillMount`. Jelikož jsem potřeboval data stáhnout ještě na serveru, moje implementovaná metoda se musela nacházet zde. V této metodě tedy zavolal mou metodu `getData`, kterou budu následně implementovat. Volání této metody jsem omezil konstrukcí if a negovanou konstantou `isFetchingDone` aby se prováděla pouze pokud data ještě nemám, abych zamezil dvojitému stahování dat, jelikož metoda `componentWillMount` se bude automaticky volat i u klienta. Poté jsem přešel na implementaci samotné bezparametrické metody `getData`.

V této metodě jsem si jako první načetl proměnné vyjadřující id produktu a typ – zda jde o nabídku, či poptávku. Načetl jsem tedy proměnné `itemId` a `type` pomocí jiné metody, která tyto informace vytáhne jako parametry routeru z `window.location.href`, což je proměnná, která vrací URL aktuální stránky. S těmito dvěma parametry jsem již mohl zavolat metodu `fetchData(type, itemId)`, kterou poskytuje interface. Aby se o této akci dozvěděl Redux, musím nejprve tuto metodu `fetchData(type, itemId)` namapovat pomocí poskytnuté metody `mapDispatchToProps` balíku React-Redux. Tato metoda slouží k vytvoření předpisu, co se má stát v případě vyslání signálu nějaké akce. V této metodě proto popíšu, co se po zavolání mé metody má stát a jaké akce se mají vyvolat. Naimplementoval jsem zde tedy to, že po zavolání

akce metody `fetchData` s jejími parametry chci zavolat metodu `fetch` ze `ServerAsyncAPI`, které v parametrech zašlu pole s požadavků, které chci odeslat na backend. V tomto konkrétním případě budu žádat data o produktu a obrázky produktu. První z nich bude volat GET request z `ItemAPI`, druhý bude volat taktéž GET request z `ItemImagesAPI`. Do obou těchto požadavků jsem předal parametry `itemId` a `type`, které poskytne původně volaná metoda `fetchData`. Poté bylo potřeba doprogramovat také všechny výše volané requesty, jelikož zatím neexistovaly. Ty jsem vytvořil v souboru api.ts, který se nachází v aktuálním adresáři.

Jako první jsem si vytvořil objekt `ItemAPI` pomocí konstruktoru `RestAPI`, ve kterém musím definovat všechny potřebné parametry. Prvním parametrem je `reducerName` – jedná se o jméno reduceru, do kterého budu ukládat přijatá data. Druhým parametrem říkám, pod jakou proměnnou budou přijatá data v reduceru dostupná, dalším parametrem je objekt `route`, který specifikuje podobu samotného požadavku. Uvnitř se nachází klíče objektu jako `url` – adresa na kterou se požadavek bude zasílat, `params` – parametry mého požadavku, zde se musí přidat například argumenty, informace o požadované paginaci, velikost paginace, poté další klíče jako `body` – tělo požadavku a `headers` – speciální hlavičky požadavku.

Jelikož požadavky na sebe můžou mít nějakou návaznost, lze dalším klíčem `events` specifikovat ještě další akce. Lze zde definovat co se má stát před samotným zavoláním požadavku nebo po něm, ale také co se má stát po úspěšném získaní dat, či naopak při chybě. Posledním klíčem je `mapper`, kde určím, která mapovací funkce bude na získaná data aplikována. Použití mapperu na data přijatá z backendu je nevyhnutelné, protože mi data přijdou z REST API ve formátu JSON. Jelikož je ale mapper naimplementován tak, že přijímá pouze objekty, před posláním dat do mapperu jsem ještě musel na všechna přijatá data aplikovat funkci z ES5 – `JSON.parse`, která se postará o převedení JSONu na Javascript objekt, se kterým již mapper umí pracovat.

Objekt `ItemAPI` jsem tedy podle výše určených parametrů nastavil, u některých atributů ale bylo nutné se řídit hodnotami, které specifikovalo REST API rozhraní backendu. Jedná se například o parametr `url`. Za zmínku stojí především nastavení akce, která se má zavolat až po mém požadavku. V objektu `events`, který je jako jeden z klíčů v objektu posílaném do konstruktoru jsem musel specifikovat, že chci `onAfter` – tedy po zavolání původního požadavku zavolat navíc ještě událost `ASYNC_PART_DONE`. Její použití a k čemu vlastně slouží vysvětlím později při implementaci `ServerAsyncAPI`, kde hraje svou velice důležitou roli. Aby vše správně fungovalo, tedy aby požadavek na data byl zaslán, získaná data předána do vnitřního úložiště aplikace a tento vnitřní stav úspěšně poslán klientovi, je nutné vyexportovat vygenerovaný `reducer`, `sagu` a `rehydrátor` z tohoto vytvořeného objektu `ItemAPI`, který se inicializací postaral o jejich automatické vytvoření. Reducer je jednoduchá funkce, která přijímá stav aplikace a akci či událost, a vrací nový stav aplikace, často modifikovaný daty poskytnuté touto akcí, či událostí. Saga je kolekce generátorů, které se starají o volání HTTP požadavků a předání akce s odpovědí dále do reduceru. Všechny tři objekty jsem vyexportoval, abych je později zaregistroval do globálních souborů, aby o nich aplikace věděla. `ItemAPI` se postará o získání všech dat o produktu

a jejich uložení do úložiště aplikace, stále mi ale chyběly případné obrázky produktu, jelikož ty se nenacházejí v databázi společně s produktem, ale jsou uloženy zvlášť v dokumentové databázi MongoDB. Bylo tak nutné ve stejném souboru api.ts vytvořit ještě jeden procesor, který bude zajišťovat získání a uložení obrázku do vnitřního úložiště aplikace, opět pomocí konstruktoru `RestAPI`, nyní jsem však objekt pojmenoval `ItemImagesAPI`. V konstruktoru jsem opět nastavil všechny potřebné parametry objektu, a i zde jsem nastavil, aby se po zavolání požadavku zavolala ještě událost `ASYNC_PART_DONE`. Vyexportoval jsem `ItemImagesAPI` vytvořený reducer, sagu a rehydrátor a ty zaregistroval.

Dále jsem musel implementovat poslední procesor, který se postará o synchronizaci těchto dvou různých požadavků. To je nutné udělat z důvodu, že server sice vyšle tyto požadavky na data, ale nečeká na případnou kladnou, či negativní odpověď z REST API a odeslal by klientovi prázdnou HTML stránku, protože data nestihla dorazit včas. Bylo tedy nutné implementovat procesor, kterým pomůžu serveru určit správný čas, kdy již může odeslat stránku klientovi – tedy poté, co oběma požadavkům přijde odpověď z backendu. K tomu poslouží objekt `ServerAsyncAPI`, který jsem inicializoval konstruktorem `AsyncAPI`. Konstruktor požaduje opět jeden objekt, který je však již o dost jednodušší než v případě `RestAPI`. Zadal jsem tedy dva nutné klíče objektu, jméno kontejneru, ve kterém se tyto asynchronní akce spouštějí – `SingleItemPage`, a pole událostí, na které se čeká. Zde se dostávám k využití akce `ASYNC_PART_DONE`. Aby server rozpoznal, že již získal všechny dostupná data, naimplementoval jsem zde čekání na tyto akce, které indikují, že jsem již data získal a je možné tedy zaslat výslednou naplněnou HTML stránku klientovi. Budu tedy čekat na dvouprvkové pole událostí, konkrétně na dvě totožné události `ASYNC_PART_DONE`, které budou vyslány, jakmile dorazí odpověď z volaných požadavků. Jako poslední provedu export a registraci v globálním souboru.

V tuto chvíli mi zbývalo pouze nadefinovat selektory v souboru selectors.ts. Selektory mi poslouží k vytáhnutí potřebných informací z vnitřního úložiště aplikace – tzv. `store`, který obsahuje celý strom zachycující kompletní stav aplikace – tzv. `state`. Důvodů pro implementaci a využití selektorů v aplikaci je hned několik. Selektory jsou v odděleném souboru, nemíchá se tak získávání dat s ostatní logikou v souboru index.tsx, který nám slouží k definici vzhledu stránky, lépe řečeno, jaké komponenty se zde budou vykreslovat a jak budou vypadat. Například pokud budu používat někde v aplikaci seznam kategorií, na jednom místě chci uživatelům dovolit vybrat ze všech kategorií, a na jiném místě chci uživatelům dovolit pouze výběr z neprázdných kategorií. Není ideální to řešit v metodě, která vykresluje tento seznam, protože to pouze znepřehledňuje kód a zvyšuje jeho složitost. Tuto jednoduchou filtraci tak můžu provést mimo `render` metodu. Pomocí dvou selektorů, každý vracející jinak upravená data. Jelikož ale v této chvíli nevím, jaká data a v jaké formě budou potřeba, nechám implementaci selektorů na později, kdy budu řešit samotné vykreslování dat.

Při code review od zkušenějšího programátora jsem byl upozorněn na možný problém při návrhu a implementaci mých metod na získání dat. Problém byl v potencionálním blokování odpovědi na serveru při čekání na události `ASYNC_PART_DONE`. Aplikace totiž musí počítat také

s chybami. Ať už ze strany aplikace, či například nedostupností samotného backendu. Při zaslání požadavku na nedostupný backend může čekání na odpověď trvat příliš dlouho, nebo je požadavek invalidní – v tomto případě taková odpověď nedorazí vůbec, což by vedlo k zaseknutí se, jelikož má implementace čeká na spuštění a odchycení události `ASYNC_PART_DONE` dvakrát za sebou, která však může dorazit až po odpovědi serveru a tato odpověď může dorazit jen jedna, nebo žádná. To by vedlo k tomu, že klient po příchodu na stránku uvidí pouze bílou stránku a server mu žádnou odpověď nepošle, jelikož stále čeká na data, ale žádné opakované volaní požadavků není implementováno, takže by došlo k zaseknutí celého procesu. To byla situace, se kterou jsem původně vůbec nepočítal, ale aby byla aplikace stabilní a takový výpadek ji nerozhodil a byla schopna v takové situaci alespoň zobrazit stránku s chybovou hláškou, či nějakou komponentu zobrazující načítání, je třeba k problému přistoupit jinak. Abych byl schopen tento neobvyklý problém vyřešit, zkušenější kolega mi poradil nahlédnout do dokumentace knihovny, se kterou tyto požadavky na data implementuji. V rozsáhlé dokumentaci Redux-Saga [9] jsem se dočetl, že přesně tyto problémy se vyskytují docela často a můj problém není tak neobvyklý, jak jsem si původně myslel. V dokumentaci je o tomto problému rovnou celá sekce, která se mu věnuje a vysvětluje jeho možná řešení. Pro můj případ se hodil ale pouze jeden. Jedná se o speciální efekt, který tuto problematiku řeší velice jednoduše, ale přesto naprosto ideálně. Je nazván Race a stará se o problematiku souběhu - čekání na několik odpovědí. Poslouží mi k řešení souběhu několika událostí a místo čekání na všechny odpovědi, u kterých není zaručeno, že vůbec dorazí, čekám pouze na první z nich, která dorazí. Byl jsem tedy nucen přepsat konstruktor `ServerAsyncAPI`. Bohužel to představovalo úpravu konstruktoru a naimplementovaní dalších metod uvnitř `ServerAsyncAPI`, což znamenalo relativně velké zdržení v postupu, kterému se nedalo nijak vyhnout.

V konstruktoru `ServerAsyncAPI` jsem tedy definoval pole o dvou prvcích, které reprezentují dva oddělené souběhy. První prvkem je událost efektu Race, která naslouchá poli definovaných akcí. Mé pole akcí se tedy skládá ze dvou prvků, souběh tedy probíhá mezi definovanými akcemi `ItemAPI.GET_SUCCESS` a `ItemAPI.GET_ERROR`. Pokud požadavek o data z `ItemAPI` bude úspěšný, zachytí se akce `GET_SUCCESS`, naopak pokud se cokoliv pokazí a odpověď nedorazí, zachycen bude `GET_ERROR`. Tento souběh jsem implementoval i pro druhý požadavek na obrázky – `ItemImagesAPI`, kde opět čekám, která z akcí `GET_SUCCESS` nebo `GET_ERROR` daného API se vrátí dříve. Poté bylo potřeba upravit samotnou třídu `AsyncAPI`. K tomuto jsem potřeboval vytvořit generátor. Díky ES6 je možné vytvářet speciální funkce zvané generátory. Ty se od klasických funkcí odlišují pomoci hvězdičky. S jejich pomocí je možné funkci uprostřed běhu přerušit, klidně i několikrát a poté v jejím běhu pokračovat, což mi dovolí spustit jiný kód uprostřed této funkce. K pozastavení provádění kódu uvnitř generátoru slouží klíčové slovo `yield`. Klasické funkce mohou přijímat parametry a mohou vracet hodnotu pomocí sekce `return`. Generátor umožňuje s každým voláním `yield` vrátit cokoliv potřebuji. V privátní generátoru `*waitForTriggerResponses` uvnitř třídy jsem si vytvořil prázdné pole `raceWinners`, do kterého budu ukládat vítěze každého závodu. Na to jsem si vytvořil anonymní funkci `saveWinner`

přijímající jeden parametr `winner`. Jediné, co tato funkce dělá je vložení proměnné `winner` do pole pomocí metody `push`. Jelikož do konstruktoru posílám pole objektů se souběhy, je nutné je v generátoru projít pomocí metody `map` zavolané na toto pole, která na každý prvek pole zavolá metodu `race` vracející tento efekt. To vše je voláno uvnitř generátoru `all` poskytnutého knihovnou Redux-Saga, který mi vrátí pole těchto souběhů, které je poté vráceno pomocí klíčového slova `yield`. Poté bylo potřeba na toto pole zavolat pro každý prvek anonymní funkci `saveWinner`, k čemuž jsem využil metodu, která zajistí její vykonání na každý objekt v poli - metodu `forEach`. Jako poslední už jsem pouze vrátil pomocí klíčového slova `yield` generátor `all`, do kterého se postupně zašlou do kanálu pomocí metody `take` všechny prvky pole `raceWinners`, což bude značit úspěšné dokončení všech potřebných souběhů, po kterém může server odeslat data klientovi.

Po implementaci jsem ještě všechny případy, které mohou nastat, otestoval a s touto vylepšenou implementací pomoci Redux-Saga Races již vše ohledně získávání dat fungovalo správně a dle očekávání.

Později během implementace samotné stránky s produktem jsem zjistil, že nemám všechna data, které potřebuji a bylo nutné tedy ještě doimplementovat získaní statistik tvůrce produktu. Abych ale získal data potřebná k zjištění statistik daného autora, bylo potřeba aby se dokončily požadavky na data produktu, ze kterého získám `id` autora a poté mohu zavolat požadavek na statistiky uživatele s daným `id`. Jedná se o další požadavek, který navíc musí čekat na dokončení předchozího požadavku na data, a proto jsem se rozhodl, že tato data bude stačit získat později až na straně klienta a nezdržovat tím celý SSR proces. U klienta to tedy bude vypadat tak, že důležitý obsah se načetl ihned, a méně důležité statistiky se budou načítat o něco déle a jakmile jsou data k dispozici, zobrazí se uživateli místo komponenty načítaní již získaná data. Požadavek byl implementován naprosto stejně jako předchozí dva požadavky na produkt a obrázky produktu, proto zde nebudu implementaci popisovat. Nakonec bylo volání akce, která vyšle požadavek na data přidáno do metody `componentDidMount`, která bude zavolána pouze na straně na klienta, jelikož se na straně serveru tato metoda vůbec nevolá.

Celá má implementace tedy funguje následovně. Při prvním vstupu klienta na stránku jsou již na serveru, běžícím na technologii Node.js zavolány požadavky na data, která jsou backendem vrácena. Data jsou poté převedeny z JSON formátu na klasické Javascript objekty. Ty dále vstupují do připravených mapperů, které mi vrátí imutabilní objekt, která již má definované i typy atributů, a tato struktura je uložena pomocí reducer funkcí. Jelikož ale není možné data odeslat klientovi v tomto tvaru, jsou opět převedeny na JSON formát a společně s HTML šablonou odeslána klientovi. Nyní přichází ke slovu rehydrátory, které se postarají o obnovení stavu vnitřního úložiště aplikace do stavu, který musí být stejný jako na serveru. Jak již bylo napsáno, data musela být před odesláním převedena na JSON, takže byly ztraceny všechny imutabilní struktury. Rehydrátory se postarají o obnovení těchto struktur ze zaslaných dat. U klienta se již vykreslí předpřipravená a daty naplněná HTML stránka.

6.4 Úloha třetí – Implementace stránky s produktem

6.4.1 Zadání úlohy

Po získání všech potřebných dat bylo mým následujícím úkolem kompletně naimplementovat stránku, která bude zobrazovat jeden produkt včetně všech jeho detailů. Stránka bude implementována dle navrženého designu a bude obsahovat galerii s fotografiemi produktu, jeho popis a také základní informace o zadavateli produktu. Navíc zde budou tlačítka pro sdílení na sociálních sítích. Dole na stránce pod samotným produktem ještě budu zobrazovat další podobné produkty, tuto funkcionalitu však bude zajišťovat oddělený kontejner, kterému budou pouze předána data z aktuálního kontejneru.

6.4.2 Rozbor problému

K implementaci stránky jsem potřeboval komponenty, starající se o správné zobrazení dat. Samotná knihovna React stojí na principu znovupoužitelnosti komponent, a proto bude veškerý obsah stránky řízen komponenty, kterým kontejner bude již pouze předávat data. Mezi požadavky aplikace bylo sdílení obsahu aplikace na sociální sítě jako Facebook, LinkedIn, Twitter. Aby se na těchto sítích zobrazovaly sdílené příspěvky správně a vypadaly atraktivně, tedy s fotkou a krátkým popisem, bylo nezbytně nutné využít také vlastní document head manager, díky kterému jsem byl schopen nastavit jednotlivé atributy hlavičky příspěvku a zajistit tím správné zobrazení obsahu. Stránka s produktem je rozdělena do několika sekcí. Nahoře se nachází název produktu a jeho typ, tedy zda se jedná o nabídku či poptávku. Pod tímto názvem je komponenta starající se o případné fotografie produktu, a dále samotný popis produktu. Na pravé straně se nacházejí tlačítka pro sdílení obsahu na sociální sítě a pod nimi karta s informacemi o tvůrci produktu a odkaz na jeho profil. Pod touto kartou, tedy na pravé straně jsem ještě zobrazil detaily samotného produktu, jako například datum vytvoření, lokaci a požadované způsoby vyrovnání. Ve spodní části stránky se nachází další samostatný kontejner starající se o zobrazení karet s podobnými parametry. Tento kontejner je ale pouze importován ze svého vlastního souboru, a proto byl při vývoji kladen důraz na jeho jednoduchou znovupoužitelnost.

6.4.3 Implementace

Z mnou řešeného předchozího úkolu jsem měl dostupná všechna data, která byla potřeba a dostat se k nim šlo pomocí již naimplementovaných selektorů. Veškerá práce ohledně vykreslování informací se již bude odehrávat pouze v souboru index.tsx, veškerá logika je tak oddělena a kód je tímto přístupem mnohem lépe čitelný a přehledný.

Veškeré vykreslování se tedy bude nacházet v metodě `render`. Jelikož se však jedná o složitou stránku, na které budu vykreslovat mnoho informací, metoda `render` by tak byla velice nepřehledná. Proto jsem vykresloval části stránky zvlášť. Toho jsem docílil tím, že jsem v metodě `render` zavolal několik dalších metod, kde každá z nich má na starost vykreslit svou

menší část. Ještě před samotnou implementací jsem však musel namapovat dříve naprogramované selektory do `props` aktuálního kontejneru. To se dělá v metodě poskytnuté knihovnou Redux `mapStateToProps` [10], která má jediný parametr `state` typu StoreStare, tedy v konkrétním případě se jedná o celý `store`, který obsahuje kompletní stav aplikace – `state`. Zde jsem namapoval všechny selektory, které budu potřebovat. Ty se musejí samozřejmě pomocí importu nejprve zavolat ze souboru selectors.ts a následně provést mapování každého z nich na proměnnou, která bude dostupná v `props`. Například proběhlo mapování `itemSelector: makeItemSelector(state)`. Proměnná `itemSelector` je datová struktura typu, uchovávající informace o produktu jako neuspořádanou množinu dvojic klíč: hodnota, známou též jako slovník. Data produktu, který budu vykreslovat, budou dostupná pod `this.props.itemSelector`. Importovaná funkce makeItemSelector již podle názvu vytvoří daný selector ve `state`, který do funkce vstupuje. Po namapování všech potřebných selektorů jsem mohl přejít na samotnou `render` metodu. `render` metoda bude vracet komponentu `React.Fragment` obalující tři samostatné vykreslovací metody, každá starající se o jinou část stránky.

Metody jsem pojmenoval `renderItemHeader`, `renderContent` a `renderSidebar`. Všechny tři metody jsou privátní a bezparametrické, jelikož jsem si všechny potřebné informace vytáhl až v samotné metodě z proměnné `this.props` a není tedy důvod posílat do metod nějaké parametry.

Začal jsem s implementací horní části stránky – metoda `renderItemHeader`, kde bude samotný název produktu a štítek označující typ produktu. Jako první jsem zde vytáhl veškerá data, které budu potřebovat. Pro získaní a vykreslení typu produktu a jeho názvu mi posloužil `itemSelector`. Pro získání této proměnné z `this.props` jsem využil nový zápis, tzv. destrukturalizace. Jedná se o zápis podporovaný od verze ES6. Umožňuje vazbu proměnných pomocí vzorové shody. Pro vytvoření štítku s typem produktu jsem využil již implementovanou komponentu Tag, která se o vykreslení štítku postará. V metodě `renderItemHeader` jsem tedy pomocí klíčového slova `return` vrátil komponentu Tag, do které pošlu dva parametry `type` a `children`. Jedná se o naší vlastní komponentu programovanou na míru přímo pro náš projekt. Parametr `type` určuje barvu, které jsou provedené pomocí konstant. Pomocí ternárního operátoru tak zjišťuji z poskytnutého selektoru `itemSelector`, zda je produkt typu `'supply'`, či `'demand'`. Na základě vrácení hodnoty ternárního operátoru je do komponenty Tag poslána buď konstanta `'spirala-primary'`, nebo `'spirala-secondary'`. Jako parametr `children` jsem do Tagu poslal právě typ produktu. Výstup z komponenty je tedy buď žlutý štítek Nabídka, nebo naopak modrý štítek Poptávka. Vykreslení samotného názvu produktu je řešeno v podstatě čistým HTML pomocí vložení proměnné nesoucí jméno produktu do JSX tagu `h4`, suplující stejnojmenný tag v HTML. Jelikož vrací metoda dvě komponenty, je nutné je obalit do jednoho rodičovského prvku. V tomto případě byl využit jako rodičovská komponenta párový HTML tag `span` z důvodu stylování obsahu mezi jeho počátečním a koncovým tagem.

Jako další jsem implementoval metodu `renderContent`, starající se o vykreslení případných obrázků produktu a samotného obsahu – popisu produktu. Samotná metoda tedy vrací pouze

další dvojici metod, každá starající se o svou část. Jedna z nich vrací komponentu pro fotografie, druhá samotný popis produktu. Obě tyto metody jsem opět musel zabalit do rodičovského prvku `React.Fragment`. Tyto dvě bezparametrické metody jsem nazval podle toho, o co se starají. Jedná se tedy o `renderItemImages` a `renderItemDescription`. K vykreslení obrázků jsem chtěl využít již implementované řešení, takže jsem se dal do hledání vhodné komponenty na veřejných repositářích stránky GitHub. Po krátkém hledání jsem již měl vybranou vhodnou komponentu starající se o vše potřebné pro naši aplikaci. Knihovnu s touto komponentu jsem jednoduše nainstaloval do projektu pomocí konzolového příkazu `yarn add react-image-gallery`, čímž došlo ke stažení knihovny do `node_modules` a vytvoření závislosti této knihovny v souboru package.json a vše bylo připraveno k použití. Tuto komponentu sloužící k zobrazení galerie fotografií jsem se rozhodl rozšířit na vlastní komponentu z důvodu přidání vlastní logiky a možné úpravy komponenty, ale hlavním důvodem bylo především opakované využití komponenty na více místech aplikace. Komponentu `ImageGallery` jsem tedy obalil vlastní třídou `Lightbox`, která rozšiřuje dědí ze třídy `React.PureComponent`. U třídy jsem nastavil objekt `defaultProps` sloužící k nastavení komponenty v případě, že tyto parametry nejsou předány rodičem, jelikož jsou označeny jako nepovinné. Objekt `defaultProps` se skládal z atributů:

- `slideDuration` – doba přechodu mezi jednotlivými snímky (v ms),
- `slideInterval` – doba zobrazení jednoho snímku (v ms),
- `infinite` – logická hodnota, zda se má po posledním snímku zobrazit znovu první,
- `showNavigation` – logická hodnota, zda mají být zobrazena tlačítka navigace,
- `thumbnailPosition` – řetězec určující umístění náhledových obrázků.

V `render` metodě mé komponenty Lightbox jsem řešil ještě logiku vykreslování. Do komponenty je nutné shora poslat seznam obrázků, které bude zobrazovat, jelikož se jedná o jediný povinný parametr komponenty. V `render` metodě tedy zkontroluji, jestli je seznam prázdný, či nikoliv. Pokud prázdný je, `render` metoda vrátí `null`, tudíž se žádná komponenta nevykreslí, jelikož není vůbec zapotřebí. Pokud ovšem seznam nějaké prvky obsahuje, vykreslím nainstalovanou komponentu ImageGallery, do které pošlu pomocí `props` všechny parametry z mé komponenty Lightbox. Pokud byl některý z parametrů poslán do `props`, je předán komponentě ImageGallery, jinak je využita výchozí hodnota z objektu `defaultProps`. Další přidanou logiku bylo potřeba naimplementovat u předávání hodnoty do property `thumbnails` – logický parametr, který rozhoduje, zda budou zobrazeny náhledové obrázky na jednom z krajů galerie. Pokud se má vykreslit obrázek pouze jeden, zakázal jsem vykreslení náhledů, jelikož je zbytečný. O veškerou zbylou logiku, i o vykreslení obrázku se již stará komponenta ImageGallery.

Co se týče vykreslení obsahu příspěvku, nejedná se o pouhé vykreslení textového řetězce. Při vytváření produktu je uživateli dovoleno využít textový editor a text tak může být pomocí stylů upraven. Výsledné HTML je před odesláním požadavku o vytvoření produktu převedeno na Markdown a ten je uložen v databázi. Pro převedení tohoto Markdown textu zpět

do HTML jsem využil další komponentu, která se již v projektu vyskytuje. Jedná se o komponentu ReactMarkdown s jediným parametrem `source`, což je samotný Markdown text. Metoda `renderContentDescription` se tedy skládá z destrukturalizace `itemSelectoru` z `this.props`, vytáhnutí proměnné pod klíčem `description` a uložení do konstanty `description`, která je následně předána komponentě, kterou metoda vrací.

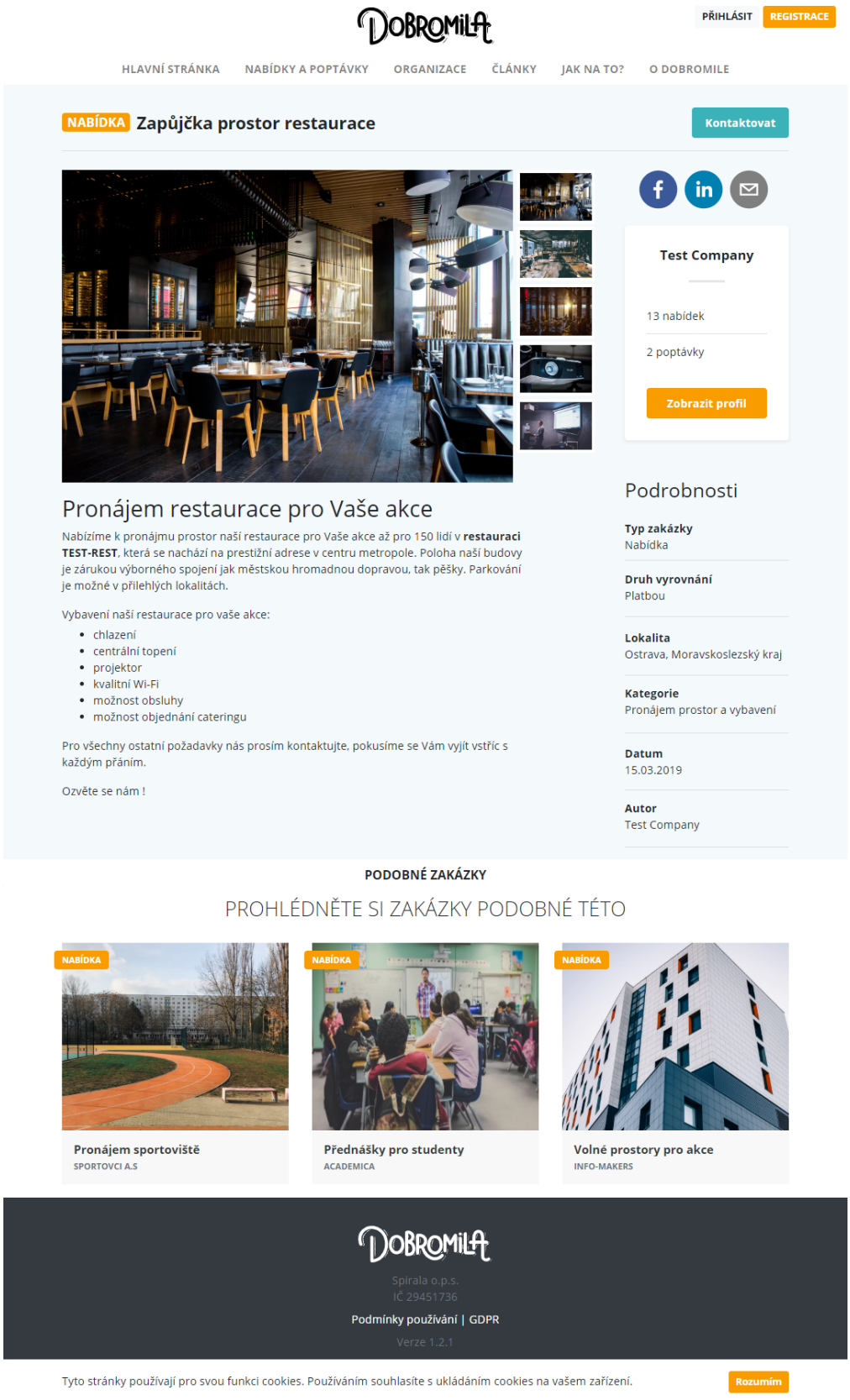
Poslední metodou starající se o zobrazení dat na pravé straně stránky byla `renderSiderbar`. Jelikož se tato metoda stará o vykreslení velkého počtu prvků, je opět přehledně rozdělena do menších sekcí, kde se každá metoda bude starat o svou nezávislou část. Samotná metoda tedy vrací zaobalenou trojici metod `renderShareButtons`, `renderAuthorCard` a `renderItemDetail`. Při implementaci metody `renderShareButtons` jsem se rozhodl metodu implementovat tak, aby vracela pouze komponentu a tlačítka tedy naimplementuji mimo kontejner v podobě komponenty. K tomuto kroku jsem se rozhodl z důvodu, že je velice pravděpodobné že tato tlačítka pro sdílení na sociální sítě se budou v aplikaci vyskytovat na více místech. V adresáři components jsem tedy vytvořil novou složku ShareButtons. Ve složce jsem založil soubor index.tsx, a začal komponentu implementovat. Třída `ShareButtons` dědí z `React.PureComponent` knihovny React. Aby bylo možné sdílet obsah stránky na které se bude komponenta vyskytovat, je nutné aby si komponenta sama zjistila, na které stránce se nachází. K uložení lokace jsem využil `state` komponenty. Tuto lokaci chci zjistit, až když je komponenta vložena do DOM, a proto jsem toto uložení lokace vložil do metody `componentDidMount`. K vložení hodnoty do `state` komponenty se využije funkce `setState`. Změna `state` komponenty znamená pro React taky přerenderování změněné komponenty, pokud však nevrátí metoda `shouldComponentUpdate` hodnotu `false`. Pokud není tato metoda programátorem implementována, je implementována implicitně s návratovou hodnotou true. Tato metoda slouží především k optimalizaci výkonu a rychlosti, proto jsem se ji rozhodl u mojí velmi malé komponenty neimplementovat.

V metodě `componentDidMount` se tedy nachází funkce `setState` s parametrem `location`, který nastavím na hodnotu `window.location.href`, která obsahuje aktuální URL adresu. Logiku tlačítek pro sdílení jsem implementoval pomocí naimportované knihovny react-share, která bude řešit zasílání požadavků na API těchto sociálních sítí, jejichž implementace není triviální a pro každou sociální síť je individuální. To byl také jeden z důvodů použití externí knihovny, která nám ušetří spoustu času, ale jsou zde i nevýhody. Mezi tu největší patří například ukončení vývoje a podpory knihovny a následná změna API některé ze sociálních sítí. To by vyústilo v nefunkční tlačítko a nutný zásah do kódu nespokojeného klienta. `Render` metoda se tedy skládá pouze z klíčového slova `return`, kde vracím `React.Fragment`, který obaluje trojici tlačítek pro sdílení na Facebook, LinkedIn a E-mail. Knihovna react-share poskytuje kromě těchto tří i několik desítek dalších sociálních sítí. Všem třem komponentám tlačítek stačilo pouze předat `url`, kterou chci v příspěvku sdílet, a nastavil jsem tak property `url` na `this.state.location`, tedy na proměnnou uloženou ve `state` této komponenty, kterou jsem po jejím vytvoření v DOM nainicializoval. Vytvořenou komponentu jsem si v kontejneru naimportoval a vložil do metody `renderShareButtons`.

Poté jsem přešel k implementaci karty autora produktu. Pro implementaci použiji komponentu Card, respektive jejího potomka ProfileCard, která upravuje styly původní komponenty Card a je připravena přesně pro mé použití. V implementované metodě jsem nainicializoval proměnné pomocí konstrukce destrukturalizace z `this.props` abych s daty mohl dále pracovat. Statistiky, které by měly být zobrazeny jsou počet nabídek, počet poptávek, a celkové hodnocení zadavatele. Statistiky obsahují také jeden neveřejný atribut a tím je počet hodnocení. Tento atribut však slouží pouze vnitřní logice komponenty, která hodnocení uživatele zobrazí pouze v případě, že jich je více než 5. Všechny tyto údaje jsou v selektoru `creatorStatsSelector`. Díky použití selektoru již nemusím data jakkoliv měnit a přidávat další kód, který nesouvisí s vykreslováním. Samotný selektor totiž upravuje data ve svém souboru, kde je nutné přepočítat hodnocení z podoby čísla 1-5 na procentuální vyjádření spokojenosti. Do komponenty tak stačí pouze předat data, o jejich vykreslení společně s doprovodným textem se postará komponenta sama. Navíc jsem se do komponenty předal také profilové informace zadavatele produktu.

Výsledné kartě již chybělo pouze tlačítko s odkazem na konkrétní profil, ale to bylo třeba naimplementovat zvlášť. K tomu jsem vytvořil další metodu `renderProfileRedirectButton`. Aby bylo možné přesměrovat uživatele na adresu v rámci aplikace, bylo třeba použít speciální komponentu Link, kterou poskytuje balík react-router-dom. Touto komponentou jsem obalil mé tlačítko a nyní zajistím správné přesměrování na profil po kliknutí, pomocí property `to` - určující cíl odkazu. Aby mohl odkaz správně fungovat, musím kromě základní cesty nastavit také dva parametry. Prvním z nich je `accountRole`, což je výčtový typ enum nabývají jedné z hodnot INDIVIDUAL, COMPANY, ORGANIZATION, ADMIN. Ten je nutné převést pomocí JavaScriptové funkce `toLowerCase` na malá písmena, jelikož odkaz počítá s malými písmeny role v adrese. Pro administrátora žádná profilová stránka neexistuje, ale taková hodnota se zde nikdy neobjeví, jelikož uživatelé s rolí admin nemůžou produkty vytvářet. Druhým parametrem je již samotné `id` tvůrce produktu. Celý tento odkaz je složen a vložen do komponenty Link a přesměrování se spustí po kliknutí na tlačítko.

Poslední věc, kterou jsem implementoval je zobrazení detailů. Metoda `renderSidebar` bude poslední implementovanou metodou v tomto kontejneru i na celé stránce. Budou zde zobrazeny informace o druhu produktu, zvolené druhy platby, město a kraj, ke kterému se produkt váže, kategorie produktu a jméno autora. Všechny tyto atributy jsou přístupné pomocí selektoru `itemSelector` pod klíčem `itemDetails`. V proměnné byl uložen seznam proměnných obsahující výše zmíněné detaily ve stejném pořadí. Jelikož potřebuji vykreslit všechny prvky seznamu, vrací se pomocí speciální konstrukce seznam již vyrenderovaných prvků. Poslouží mi k tomu metoda `map`, jež na každý prvek seznamu, na který je volána, provede předanou funkci. Každý prvek seznamu jsem předal metodě `renderItem`, která se postará o jeho vykreslení a kterou následně implementuji. Metoda přijímá jediný parametr `item`, ze kterého si vytáhnu název klíče i hodnotu a uložím si je do proměnných `title` a `description`. Dále v metodě již jen vracím v `React.Fragment` zabalenou dvojici párových značek `div`, kde každý z nich obaluje jednu z proměnných `title` a `description`.



Obrázek 2: Kompletní stránka produktu

6.5 Úloha čtvrtá – Vytvoření ElasticSearch query

6.5.1 Zadání úlohy

Jelikož na vyhledávání dat používáme v projektu místo klasické SQL databáze ElasticSearch, je nutné vytvořit vyhledávací dotaz přesně pro tento vyhledávací engine. Konkrétně jde o dotaz na vyhledávání podobných záznamů k jednomu konkrétnímu produktu.

6.5.2 Rozbor problému

Použití vyhledávacího enginu ElasticSearch přináší oproti vyhledávání v klasické SQL databází především benefit rychlosti. ElasticSearch byl navržen a také postaven tak, aby byl schopen vyhledávat v miliardách záznamů, a to zcela spolehlivě a ke všemu mnohem rychleji než klasické SQL databáze. Mým úkolem bylo napsat ElasticSearch query sloužící k vyhledávání nejpodobnějších záznamů k danému produktu. Před samotným psaním dotazu mě čekalo čtení dokumentace a učení, jak to vlastně funguje.

6.5.3 Implementace

Protože jsem se s vyhledávacím a analytickým enginem ElasticSearch ještě nesetkal, bylo nutné před tím, než začnu sestavovat query - dotaz, si o tom jak ElasticSearch vlastně funguje přečíst v dokumentaci. Již po krátké době jsem pochopil, že tato technologie je opravdu velice odlišná od klasických SQL Select dotazů. V ElasticSearch se místo tabulek používá typ a místo řádků se ukládají dokumenty. Přidání záznamu do databáze ElasticSearch je vcelku jednoduché, kromě samotného vložení si již vše obstarává sám. Nový záznam je potřeba kvůli extrémně rychlému vyhledávání zaindexovat. Implementace ElasticSearch jde až na samou hranici optimalizace, proto je změna záznamu a úprava například jména produktu pro ElasticSearch velký problém. V SQL databázi bychom použili klasický DML Update, kde hodnotu jednoduše upravíme. Pro ElasticSearch je taková změna velice obtížná, a proto je pro něj lepší starý záznam smazat a raději vložit nový upravený záznam znovu. Nicméně po chvíli jsem se dostal k sekci Wildcard Query [11]. Tyto dotazy hledají a spojují dokumenty, které splňují zástupný výraz - tzv. `wildcard expression`. Mezi podporované wildcards patří * - jakákoliv sekvence znaků, včetně prázdné sekvence, dále ? - reprezentující jakýkoliv jeden znak. Tento dotaz ale může být docela pomalý z důvodu nutného procházení velkým množstvím možností. Dalším omezením je, že výraz by neměl začínat znakem * ani ? - to by zapříčinilo, že hledání by bylo extrémně pomalé, jelikož by musel ElasticSearch hledat v podstatě jakoukoliv kombinaci pouze končící daným výrazem. Dotaz v podstatě nic dalšího nepodporuje a typ dotazu Wildcard Query se tak ukázal jako slepá ulička.

Mnohem více nadějným kandidátem se tak stalo po delším hledání v dokumentaci More Like This Query - MLT Query zkráceně [12]. Tento dotaz hledá dokumenty, které se podobají množině daných dokumentů. MLT vybere reprezentativní pojmy ze vstupních dokumentů, zfor-

muluje dotaz na základě těchto pojmů, vykoná tento dotaz a vrátí výsledek. Samozřejmě, že nejpodobnějším dokumentem k zadanému dokumentu je dokument sám, a to z důvodu, že dosahuje nejvyššího `tf-idf` vypočítaného podle bodovacího vzorce Lucene [13]. MLT dotaz tedy jednoduše získá text ze vstupních dokumentů a provede pomocí nich selekci N dokumentů s nejvyšším `tf-idf` skóre. Jediným nutným parametrem tohoto dotazu je tak parametr `like`, určující vstupní množinu dokumentů. ElasticSearch ale navíc podporuje několik velice užitečných rozšíření tohoto dotazu, které mi umožnily ještě lépe optimalizovat tento dotaz. Ty se dělí na tři základní skupiny parametrů:

- vstupní dokumenty – specifikace vstupních dokumentů,
- výběr pojmů – specifikace vybíraných pojmů a
- formování dotazu – specifikace modifikací dotazu.

Prvním typem jsou tedy parametry specifikující vstupní dokumenty a detaily toho, co je hledáno. Text, který se má extrahovat ze vstupních dokumentů je možné buď explicitně určit pomocí parametru `fields`, jinak budou použity všechny dostupné texty dokumentu. Zajímavým parametrem je parametr `unlike`. Tento parametr je využit ve spojení s parametrem `like` a lze jím určit, aby se nevybíraly určité pojmy či celé dokumenty. Například pokud bych chtěl získat dokumenty podobné, či nějak spojené se slovem "Jablko", ale nezajímají mě dokumenty zmiňující "Jablečný koláč", využil bych právě tuto konstrukci.

Další skupinou jsou parametry specifikující samotné pojmy. Zde je výběr už o dost širší, nastavit lze opravdu hodně pravidel, a popíši pouze pár z nich. Parametr `max_query_terms` slouží k určení maximálního počtu pojmů, které budou z dokumentu vybrány. Čím větší číslo je vybráno (výchozí hodnota je 25), tím přesnější bude i výsledek, avšak na úkor času vykonání dotazu. Další parametr `min_term_freq` určuje minimální počet výskytů pojmu, pod který je pojem ze vstupních dokumentů ignorován. To můžou být například slova, která se v textu vyskytují spíše náhodně, se samotným obsahem příliš nesouvisí a jsou zde jednou až dvakrát. Poslední parametr `stop_words` umožňuje definovat množinu slov, která jsou považována za nezajímavá a budou ignorována.

Poslední skupinou jsou parametry pro formování dotazu. Opět zmíním pouze několik parametrů. Parametrem `minimum_should_match` můžu nastavit hranici minimální shody všech vybraných pojmů se vstupními dokumenty. Výchozí hodnota je 30%. Další parametr`boost` umožňuje nastavit hodnotu zvýšení důležitosti pro celý dotaz. Lze také nastavit `boost` daného dotazu pro každý atribut zvlášť. Užitečným příkladem by bylo například nastavení `boost` pro atribut `name` na hodnotu 3.0 a `description` na hodnotu 1.0, čímž bych řekl, že shody v názvu produktu jsou pro mě třikrát důležitější a relevantnější něž shody s jeho popisem. To by však vyžadovalo přepsání mého MLT query na dvě menší MLT Query pod rodičovským `query dis_max`. Pro začátek jsem se rozhodl pro jednodušší variantu, kterou lze v případě nedostatečné přesnosti hledání podobných produktů kdykoliv přepsat. Po testování dotazu pomocí vývojářské konzole

v nástroji Kibana jsem měl hotové své první ElasticSearch query, které ovšem nemohlo být takto přímo použito. Z důvodu zabezpečení nemohl odesílat tyto dotazy na REST API databáze ElasticSearch přímo frontend, ale bylo nutné dotaz přeposílat společně s parametrem `_id`, které se bude vždy měnit podle toho, ke kterému produktu se budou hledat podobné záznamy na REST API našeho backendu, který dotaz vykonal a výsledek hledání přeposlal zpět. Neobsahoval tak žádnou logiku navíc, ale fungoval pouze jako prostředník předávající výsledek, díky čemuž mohla být komunikace s databází ElasticSearch povolena pouze s backendem. Dotaz tedy ještě musel být implementován v jazyce Java, to však nebyl žádný problém, jelikož je tento způsob přímo podporován vytvořenou knihovnou.

```
POST demand,supply/_search
{
   "query": {
            "more_like_this": {
               "fields": ["name", "description"],
               "like": [
                 {
                  "_id": "01bbf4e8-562d-40ee-9853-fcf3098ecdf1"
                 }
               ],
               "min_term_freq": 3,
               "min_doc_freq": 3,
               "max_query_terms": 30
             }
           }
}
```

Výpis 3: Příklad MLT ElasticSearch Query

7 Hodnocení znalostí a dovedností potřebných v průběhu praxe

7.1 Uplatněné znalosti a dovednosti

Na praxi jsem mohl uplatnit mnoho znalostí nabytých studiem na vysoké škole. Mnoho podkladů dodaných od zákazníka, pro kterého byla aplikace vytvářena mi bylo velice známých z předmětu Úvod do softwarového inženýrství. Také jsem si prakticky vyzkoušel fungování agilní metodiky Scrum, se kterým jsem měl prozatím z výše uvedeného předmětu zkušenosti pouze teoretické. Při programování frontend části aplikace v jazyce TypeScript pro mě bylo velice přínosné především absolvování předmětů Programování II a Algoritmy II. Mnoho zkušeností a znalostí jsem při návrhu systému a jeho implementaci využil také z předmětů Vývoj informačních systému a Databázové a informační systémy. Ke spravování verzí během celého vývoje aplikace byl využit nástroj na správu verzí GIT, který byl v základu představen a používán při vývoji projektu v předmětu Tvorba aplikací pro mobilní zařízení II. Částečně mi pomohl i předmět Uživatelská rozhraní, i když se v něm programovalo v jiném jazyce, mnoho poznatků jsem z něj při tvorbě aplikace využil.

7.2 Chybějící znalosti a dovednosti

I přes využití mnoha technologií a znalostí získaných během studia jsem narazil na oblasti, ve kterých jsem neměl příliš zkušeností. Především jsem se stále učil nové věci ohledně knihovny React, se kterou jsem neměl příliš mnoho zkušeností a čtením dokumentace jsem strávil opravdu mnoho času. Ocenil bych zařazení verzovacího systému GIT a jeho pokročilejších funkcí do některého z povinných předmětů bakalářského studia, jelikož základní dovednost mi nestačila. S tím lehce souvisí i moje první setkání s vývojem softwaru v týmu, který jsem si během studia nikde neměl možnost vyzkoušet. Celý frontend aplikace byl psaný v jazycích TypeScript a JavaScript, k nimž se bohužel během studia neváže žádný povinný předmět a musel jsem se v něm učit až během praxe.

8 Závěr

Odbornou praxi ve společnosti Tieto Czech s.r.o hodnotím velice kladně a beru ji jako velice cennou část mého bakalářského studia. Během praxe jsem měl možnost pracovat a velice dobře porozumět knihovnám a jazyku JavaScript, které převládají při vývoji dnešních moderních webových aplikací. Velice si cením zkušenosti s prací v týmu a na vlastní kůži jsem si vyzkoušel i fungování metodiky Scrum. Díky spolupráci se zkušenými kolegy programátory jsem nabyl mnoho nových znalostí a zlepšil i své postupy při profesionálním vývoji softwaru od návrhu až po jeho nasazení do provozu. Cením si také zkušenosti s prací ve velké IT firmě, mezi které se Tieto určitě řadí. Jsem rád, že jsem mohl pracovat na projektu, který má přesah i mimo firmu Tieto, neboť je to aplikace, která by měla pomáhat neziskovým organizacím, a proto je také veřejně dostupná na adrese www.dobromila.net. Absolvování této praxe mi dalo mnoho zkušeností a hlavně rozhled v relativně krátkém čase a věřím, že všechny tyto věci využiji jak při svém navazujícím magisterském studiu, tak také v budoucím zaměstnání.

Literatura

[1] *Historie Tieta* [online]. [cit. 2019-02-01].
Dostupné z: https://campaigns.tieto.com/tieto50/mobile/

[2] *Dokumentace knihovny React* [online]. [cit. 2019-02-01].
Dostupné z: https://reactjs.org/docs

[3] *Jazyk TypeScript* [online]. [cit. 2019-02-03].
Dostupné z: https://www.typescriptlang.org/

[4] *Metody životního cyklu v knihovně React* [online]. [cit. 2019-02-03].
Dostupné z: http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

[5] *React PureComponent* [online]. [cit. 2019-02-08].
Dostupné z: https://reactjs.org/docs/react-api.html#reactpurecomponent

[6] *Dokumentace knihovny Redux* [online]. [cit. 2019-02-08].
Dostupné z: https://react-redux.js.org

[7] *React Portals* [online]. [cit. 2019-02-16].
Dostupné z: https://reactjs.org/docs/portals.html

[8] *React Fragment* [online]. [cit.2019-02-16].
Dostupné z: https://reactjs.org/docs/react-api.html#reactfragment

[9] *Dokumentace knihovny Redux-Saga* [online]. [cit. 2019-02-28].
Dostupné z: https://redux-saga.js.org

[10] *Redux mapStateToProps* [online]. [cit. 2019-02-28].
https://react-redux.js.org/using-react-redux/connect-mapstate

[11] *Wildcard Query* [online]. [cit. 2019-03-15].
Dostupné z: https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-wildcard-query.html

[12] *More Like This Query* [online]. [cit. 2019-03-15].
Dostupné z: https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-mlt-query.html

[13] *Lucene scoring vzorec* [online]. [cit. 2019-03-16].
Dostupné z: https://lucene.apache.org/core/7_4_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html